# Routing

HTML5 introduced the history.pushState() and history.replaceState() methods, which allow you to add and modify history entries, respectively.

Thanks to pushState, we can make our in-app URL paths look the way we want them to look, e.g. localhost:3000/crisis-center. Our in-app URLs can be indistinguishable from server URLs. Modern HTML 5 browsers were the first to support pushState which is why many people refer to these URLs as "HTML 5 style" URLs.

We must add a <base href> element tag to the index.html to make pushState routing work. The browser also needs the base href value to prefix relative URLs when downloading and linking to css files, scripts, and images.

Add the base element just after the <head> tag. If the app folder is the application root, as it is for our application, set the href value in index.html exactly as shown here.

<base href="/">

## Configure and add the router

# Two-way Binding with NgModel

When developing data entry forms, we often want to both display a data property and update that property when the user makes changes.

The [(ngModel)] two-way data binding syntax makes that easy. Here's an example:

<input [(ngModel)]="currentHero.firstName">

[()] = BANANA IN A BOX

To remember that the parentheses go inside the brackets, visualize a banana in a box.

Alternatively, we can use the canonical prefix form:

<input bindon-ngModel="currentHero.firstName">

We could have achieved the same result with separate bindings to the <input> element's value property and input event. That’s however cumbersome.

<input [value]="currentHero.firstName" (input)="currentHero.firstName=$event.target.value" >

That ngModel directive hides these onerous details behind its own ngModel input and ngModelChange output properties.

<input [ngModel]="currentHero.firstName" (ngModelChange)="currentHero.firstName=$event">

*The ngModel input property sets the element's value property and the ngModelChange output property listens for changes to the element's value. The details are specific to each kind of element and therefore the NgModel directive only works for elements, such as the input text box, that are supported by a ControlValueAccessor. We can't apply [(ngModel)] to our custom components until we write a suitable value accessor.*

We shouldn't have to mention the data property twice. Angular should be able to capture the component’s data property and set it with a single declaration — which it can with the [( )] syntax:

<input [(ngModel)]="currentHero.firstName">

### Syntactic suger[(x)]

[(x)] is just syntactic sugar for a property binding and an event binding:

[x]="someParentProperty" (xChange)="someParentProperty=$event"

To achive the following code:

<my-comp [(myText)]="testString"></my-comp>

In the component for my-comp must have an myText property and an myTextChange property(which is a EventEmitter).

**export class** MyComp {  
 @Input() **myText**: **string**;  
 @Output() **myTextChange**: EventEmitter<**string**> = **new** EventEmitter();

*//Notify parent of changes, whenever the value of myText changes, emit an event.* onChange(newMyText:**string**) {  
 **this**.**myTextChange**.emit(newMyText);  
 }  
}

### Aliasing input/output properties

Sometimes we want the public name of an input/output property to be different from the internal name.

This is frequently the case with [attribute directives](#_Attribute_directives). Directive consumers expect to bind to the name of the directive. For example, when we apply a directive with a myClick selector to a <div> tag, we expect to bind to an event property that is also called myClick.

<div (myClick)="clickMessage=$event">click with myClick</div>

However, the directive name is often a poor choice for the name of a property within the directive class. The directive name rarely describes what the property does. The myClick directive name is not a good name for a property that emits click messages.

Fortunately, we can have a public name for the property that meets conventional expectations, while using a different name internally.

@Output('myClick') clicks = new EventEmitter<string>(); // @Output(alias) propertyName = ...

Or alternative:

@Directive({

outputs: ['clicks:myClick'] // propertyName:alias

})

# Directives

There are three kinds of directives in Angular:

* Components
* Structural directives
* Attribute directives

A Component is really *a directive with a template*. It's the most common of the three directives and we tend to write lots of them as we build applications.

Structural directives can change the DOM layout by adding and removing DOM elements. **NgFor** and **NgIf** are two familiar examples.

An Attribute directive can change the appearance or behavior of an element. The built-in **NgStyle** directive, for example, can change several element styles at the same time.

We don't need any directive to simply set the background color. We can set it with the special Style Binding like this:

<p [*style.background*]="'lime'">I am green with envy!</p>

## Attribute directives

An attribute directive minimally requires building a controller class annotated with @Directive, which specifies the selector identifying the attribute associated with the directive. The controller class implements the desired directive behavior.

An example highlight.directive.ts:

**import** {Directive, ElementRef} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **constructor**(el: ElementRef){  
 el.**nativeElement**.**style**.**backgroundColor** = **'yellow'**;  
 }  
}

We need the ElementRef to inject into the directive's constructor so we can access the DOM element.

@Directive requires a CSS selector to identify the HTML in the template that is associated with our directive. The CSS selector for an attribute is the attribute name in square brackets. Our directive's selector is *[myHighlight]*.

We export `HighlightDirective` to make it accessible to other components.

**import** {Component} **from "@angular/core"**;  
**import** {HighlightDirective} **from "./highlight.directive"**;  
@Component({  
 **selector**: **'my-app'**,  
 **directives**: [HighlightDirective],  
 **template**: **'<h1>My First Attribute Directive</h1><p myHighlight>Highlight me!</p>'**})

Angular creates a new instance of the directive's controller class for each matching element, injecting an Angular ElementRef into the constructor. ElementRef is a service that grants us direct access to the DOM element through its nativeElement property. That's all we need to set the element's background color using the browser DOM API.

### Respond to user action

Our directive should be able in response to a user action. We apply the @HostListener decorator to methods which are called when an event is raised.

@HostListener('mouseenter') onMouseEnter() {/\* . . . \*/}

@HostListener('mouseleave') onMouseLeave() {/\* . . . \*/}

**import** {Directive, ElementRef, HostListener} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **constructor**(**private** el: ElementRef){}  
 @HostListener(**'onmouseenter'**)  
 **private** onMouseEnter(){  
 **this**.hightlight(**'yellow'**);  
 }  
 @HostListener(**'onmouseleave'**)  
 **private** onMouseLeave(){  
 **this**.hightlight(**'null'**);  
 }  
 **private** hightlight(color:**string**) {  
 **this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

### Configure the directive with binding

We should set the color externally with a binding like this:

<p [myHighlight]="color">Highlight me!</p>

We'll extend our directive class with a bindable input highlightColor property and use it when we highlight text.

**import** {Directive, ElementRef, HostListener, Input} **from "@angular/core"**;  
@Directive({  
 **selector**: **'[myHighlight]'**})  
**export class** HighlightDirective {  
 **private defaultColor**:**string** = **'red'**;  
 **constructor**(**private** el: ElementRef){}

*//@Input(alias) alias the highlightColor property with myHighlight*  
 @Input(**'myHighlight'**) **highlightColor**:**string**;  
 @HostListener(**'onmouseenter'**)  
 **private** onMouseEnter(){  
 **this**.hightlight(**this**.**highlightColor** || **this**.**defaultColor**);  
 }  
 @HostListener(**'onmouseleave'**)  
 **private** onMouseLeave(){  
 **this**.hightlight(**'null'**);  
 }  
 **private** hightlight(color:**string**) {  
 **this**.el.**nativeElement**.**style**.**backgroundColor** = color;  
 }  
}

Update the template:

<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>

<div>

<input type="radio" name="colors" (click)="color='lightgreen'">Green

<input type="radio" name="colors" (click)="color='yellow'">Yellow

<input type="radio" name="colors" (click)="color='cyan'">Cyan

</div>

<p [myHighlight]="color">Highlight me!</p>

### Bind to a second property

Let's allow the template developer to set the default color, the color that prevails until the user picks a highlight color. We'll add a second input property to HighlightDirective called defaultColor:

@Input() **set** defaultColor(colorName: **string**){  
 **this**.**defaultColor** = colorName || **this**.**defaultColor**;  
}

In html-template:

<p [myHighlight]="color" [defaultColor]="'violet'">

Highlight me too!

</p>

# Words

**cumbersome** 英 **['kʌmbəsəm]**  美 **['kʌmbərsəm]**  **adj.笨重的；不方便的**

**cumber** 英 ['kʌmbə]   美 ['kʌmbə]  v.**拖累；妨碍** n.**妨碍；累赘**

**onerous** 英 ['əʊnərəs]  美 ['ɑːnərəs] adj.**繁重的**